Московский государственный технический университет

имени Н.Э. Баумана

Факультет «Информатика и вычислительная техника»

Кафедра «Компьютерные системы и сети»

Т.Н. Ничушкина, В.В. Гуренко

**Разработка программ модульной структуры**

Учебно-методическое пособие по выполнению Домашнего задания № 3 первого семестра по дисциплине «Информатика»

Москва

(С) 2023 МГТУ им. Н.Э. БАУМАНА

УДК 004.432

Рецензент: доцент кафедры ИУ7 к.т.н. Новик Н.В..,

Т.Н. Ничушкина, В.В. Гуренко

Разработка программ модульной структуры. Учебно-методическое пособие по выполнению Домашнего задания № 3 по дисциплине «Информатика» **-** М.: МГТУ имени Н.Э. Баумана, 2019. 36 с.

Учебное пособие содержит теоретические сведения об особенностях разработки программ модульной структуры в языке С++.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по программе бакалавриата направления «Математика и компьютерные науки»

*Рекомендовано Учебно-методической комиссией НУК «Информатика и системы управления» МГТУ им. Н.Э. Баумана*

Ничушкина Татьяна Николаевна, Владимир Викторович Гуренко

Разработка программ модульной структуры.

**Учебно-методическое пособие по выполнению домашнего задания № 3 по дисциплине «Информатика»**

© Т.Н. Ничушкина, В.В.Гуренко, 2023

© МГТУ имени Н.Э. Баумана, 2023

Оглавление

[Введение 4](#_Toc142580319)

[1. Теоретическая часть 5](#_Toc142580320)

[1.1. Модульное программирование 5](#_Toc142580321)

[**1.1.1.** **Подпрограммы в С/С++** 5](#_Toc142580322)

[**1.1.2.** **Модули С/С++** 9](#_Toc142580323)

[**1.1.3.** **Средства создания универсальных подпрограмм. Указатель на функцию** 13](#_Toc142580324)

[1.1.3.1. Применение указателя на функцию как параметра. 13](#_Toc142580325)

[1.1.3.2. Пример использования указателя на функцию. 17](#_Toc142580326)

[1.2. Обработка списковых структур данных 20](#_Toc142580327)

[1.3. Тестирование 30](#_Toc142580328)

[2. Практическая часть. 33](#_Toc142580329)

[2.1. Порядок выполнения задания 33](#_Toc142580330)

[2.2. Требования к оформлению. 33](#_Toc142580331)

[2.3. Требования к защите 34](#_Toc142580332)

[2.4. Пример выполнения задания 35](#_Toc142580333)

[2.5. Примеры вариантов домашнего задания 49](#_Toc142580334)

[3. Контрольные вопросы 52](#_Toc142580335)

[Заключение 53](#_Toc142580336)

[Список литературы 54](#_Toc142580337)

[Приложение. Пример титульного листа. 55](#_Toc142580338)

# Введение

Изящное и технически грамотное написание компьютерных программ предполагает не только хорошее владение средствами их разработки, но и достаточно развитое алгоритмическое мышление. Сложившаяся на протяжении десятилетий практика обучения программированию говорит о том, что именно недостаток алгоритмического мышления – основная причина неудач студентов в процессе освоения программирования.

Алгоритмическое мышление как умение выстраивать логически безупречную последовательность действий на пути к решению задачи, можно и нужно развивать. Опыт показывает, что одним из наиболее действенных способов достижения этого является проработка алгоритмов ряда небольших, но полезных программ, приводящая к формированию базы приемов программирования. Таких приемов сравнительно немного, но их накопление, обобщение и умение осознанно применять на практике позволяет студентам научиться писать программы.

В настоящем учебно-методическом пособии представлены средства разработки универсальных программ с использованием для решения задач модульного подхода, а также рассмотрены средства и приемы программирования, которые используются при работе с динамическими структурами данных, позволяющие оптимизировать использование оперативной памяти для хранения и обработки данных.

# Теоретическая часть

**Цель работы:** изучение средств разработки универсальных подпрограмм и особенностей использования этих средств при программировании. Освоение приемов работы с динамическими структурами данных.

## Модульное программирование

Для улучшения эффективности программ в языках высокого уровня были разработаны средства модульного программирования, предусматривающие использование подпрограмм и модулей.

***Подпрограмма* –** это относительно самостоятельный фрагмент программы, соответствующим образом оформленный и снабженный именем.

В зависимости от способов описания, формирования результата и вызова различают подпрограммы двух видов: процедуры и функции.

*Процедуры* предназначены для выполнения некоторых действий, например, печати строки, а *функция* – позволяет получить задаваемую скалярную величину, которую возвращает в качестве результата.

С\С++ декларированы как языки *функционального* программирования. Поэтому в С\С++ нет понятия процедуры, как средства языка. Однако существует возможность создания функций, которые не возвращают значения и реализуют действия, свойственные процедурам.

### **Подпрограммы в С/С++**

Каждая программа на С/С++ обязательно должна включать единственную главную функцию с именем main. Кроме нее в программу может входить произвольное количество функций, выполнение которых прямо или косвенно инициируется функцией main.

Функции можно описывать в любом месте программы и даже в другом файле, но только не внутри другой функции.

Различают ***объявление*** и ***описание*** функций.

*Описание функции* состоит из заголовка, при необходимости используемого в качестве прототипа, и собственно тела функции.

**<Тип результата> <Имя подпрограммы> ([<Список параметров>])**

**{ [< Объявление локальных переменных и констант >]**

**<Операторы>**

**}**

*Объявление функций* предполагает включение в начало программы *прототипов* (заголовков) всех используемых функций, тогда описания функций могут быть приведены в любом порядке. Если прототипы функций в начале программы не помещать, то все функции должны быть обязательно описаны до своего явного вызова.

**Пример объявления и описания подпрограммы:**

int max(int a, int b); // Объявление подпрограммы (прототип)

int max(int a, int b) // Описание подпрограммы

{

if (a>b) return a;

else return b;

}

Функция может получать данные двумя способами:

а) неявно – с использованием видимых в других функциях переменных;

б) явно – через параметры.

При отсутствии параметров скобки оставляют пустыми или список можно заменить служебным словом void (int print(void)) или опустить (int print()).

**Формальные и фактические параметры.** *Формальными*называют параметры, определенные в заголовке описания функции. При перечислении для каждого формального параметра помимо имени задают тип, например:

float max (**float** a, **char** b) {…}

*Фактическими*называют параметры, задаваемые при вызове функции. В математике такие параметры называют аргументами. В качестве аргументов можно использовать литералы, поименованные константы, переменные и выражения.

Совокупность формальных параметров определяет *сигнатуру функции.* Сигнатура функциизависит от количества параметров, их типа и порядка размещения в спецификации формальных параметров.

При задании аргументов следует помнить, что формальные и фактические параметры, относящиеся к одной функции, должны совпадать: по количеству параметров, по их типу и по порядку следования.

**Возврат результатов через параметры функции (процедуры).**

Существует два способа передачи данных в подпрограммы через *параметры.* Первый из них известен как «передача параметров по значению». В этом случае в подпрограмму передаются *копии фактических* *параметров*, и никакие изменения этих копий не возвращаются в вызывающую программу.

Второй способ называют «передачей параметров по адресу». При использовании этого метода в подпрограмму передаются *адреса фактических параметров*, соответственно, все изменения этих параметров в подпрограмме происходят с переменными основной программы. *По умолчанию в языках С/С++ применяется передача по значению.*

Если подпрограмма меняет передаваемое значение, и это измененное значение надо вернуть в вызывающую программу, то используют один из двух способов:

а)в качестве параметра *применяют указатель* ипри вызове передают адрес;

б)в качестве параметра *применяют ссылку* и при вызове просто указывают имя параметра.

В обоих вариантах подпрограмма получает *копию адреса*, по которому надо записать возвращаемое значение. Например:

а) возврат значения с использованием параметров-указателей:

описание функции:

void prog(int a, int \*b){ \*b = a; }

вызов функции:

prog(c, &d);

б) возврат значения с использованием параметров-ссылок:

описание функции:

void prog(int a, int &b){ b = a;}

вызов функции:

prog(c, d);

Если наоборот, надо *запретить изменение параметра*, переданного адресом, то его объявляют с описателем const, например:

int prog2(const int \*a) {…}

**Оператор возврата значения функции.**

Если описывается подпрограмма-функция, которая возвращает в вызывающую программу формируемое значение, то в теле функции обязательно наличие оператора возврата return, формирующего и передающего это значение, например:

int max(int a,int b)

{

if (a>b) return a;

else return b;

}

Вызовфункции может выглядеть, например, так:

k=max(i,j); **.**

В соответствии со стандартом, если функция, возвращающая значение, не использует оператора return**,** то в вызывающую программу возвращается случайное значение (называемое *мусором*).

При необходимости функция может содержать несколько операторов возврата return, как в примере выше.Встретив любой из них, функция формирует возвращаемое значение и передает управление в конец подпрограммы.

Если функция не объявлена со спецификацией void, то ее можно использовать в качестве операнда в любом выражении, например:

K=max(a,b)/max(c,e).

Следует отметить, что оператор return в схеме алгоритма изображается как обычный блок *процесс*, так как он только формирует результат по правилам С/С++ и передает управление на конец подпрограммы. А уже после возврата управления в точку завершения подпрограммы, последняя выполняет стандартные действия по возврату управления вызвавшей ее программе или подпрограмме.

### **Модули С/С++**

Среды Visual Studio и Qt Creator позволяют создавать и отлаживать программы, использующие не только стандартные, но и пользовательские библиотеки подпрограмм – *модули*.

Модуль C/C++ представляет из себя коллекцию ресурсов программы: константы, переменные, типы и подпрограммы. Обычно модуль включает два исходных файла: заголовочный файл с расширением «.h» и файл реализации с расширением «.cpp».

*Заголовочный файл* играет роль интерфейсной секции модуля. В него помещают объявление экспортируемых ресурсов модуля:

* прототипы (заголовки) процедур и функций,
* объявление переменных, типов и констант.

Заголовочный файл подключают командой #include "<Имя модуля>.h", записываемой в файле реализации программы или другого модуля, если они используют ресурсы описываемого модуля.

*Файл реализации* представляет собой секцию реализации модуля. Он должен содержать команды подключения используемых модулей, описания экспортируемых процедур и функций, а также объявление внутренних ресурсов модуля. Файл реализации подключается командой #include "<Имя модуля>.cpp".

При создании первый (главный) файл проекта уже содержит заготовку основной функции программы – функции main(). Для создания файлов модуля и добавления их к проекту можно использовать два пути:

1. Вновь вызвать многошаговый Мастер заготовок. Это делается с использованием команды меню File/New. Выполнение этой команды при открытом проекте вызовет открытие окна Мастера заготовок на вкладке Files, на которой необходимо выбрать тип файла, добавляемого к проекту и задать его имя.
2. Использовать команду меню Project/Add New Items. Выполнение этой команды также вызовет открытие окна Мастера заготовок, с помощью которого необходимо выбрать тип файла, добавляемого к проекту и задать его имя.

Следует отметить, что модули С/С++, в отличие от модулей некоторых других языков программирования, хранятся не в откомпилированном виде, а в исходных текстах. Именно поэтому их подключают специальными директивами препроцессора, чтобы при компиляции программы их исходные тексты были включены в исходный текст программы и обработаны компилятором вместе с программой.

**Пример 1.** Разработать модуль для реализации нахождения НОД двух целых чисел.

*Файл mod.h:*

#ifndef modh\_20180810

#define modh\_20180810

int nod(int a,int b);

#endif

Для оформления файлов модуля используются дополнительные команды препроцессора

*#ifndef <Идентификатор>,*

*#define <Идентификатор>,*

*#endif*

Так, команды препроцессора в примере

#*ifndef modh\_20180810,*

*#define modh\_20180810,*

*#endif****,***

в файле mod.h позволяют исключить повторную компиляцию текста программы при многократном подключении заголовочного файла за счет определения для него уникального идентификатора. Чтобы определить уникальный идентификатор, который точно не встречается в других библиотеках, можно использовать следующую схему, которая и применена в примере для mod.h:

**<ИМЯФАЙЛА><РАСШИРЕНИЕ>\_<ГОД><МЕСЯЦ><ДЕНЬ (создания)>.**

В нашей программе это modh\_20180810, который включает как имя модуля, так и дату его создания.

*Файл mod.cpp: // файл реализации*

#include "mod.h"

int nod(int a,int b)

{

while (a!=b)

if (a>b) a=a-b;

else b=b-a;

return a;

}

*Файл main.cpp: // файл основной программы*

#include <locale.h>

#include <stdio.h>

#include <conio.h>

#include "mod.h"

int main()

{

setlocale(0,"russian");

int a=18,b=24,c;

c=nod(a,b);

printf("Наибольший общий делитель=%d\n",c);

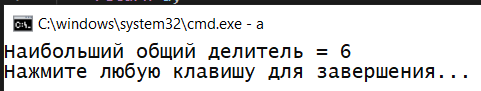
puts("Нажмите любую клавишу для завершения...");

\_getch();

return 0;

}

**Результат работы программы:**



На рисунке 1 приведена схема взаимодействия заголовочного и исполнительного файлов модуля mod.



Рисунок 1 – Схема взаимодействия файлов программы при модульном программировании.

### **Средства создания универсальных подпрограмм. Указатель на функцию**

При проектировании сложной программы разработчик стремится создать как можно более универсальные подпрограммы, чтобы сократить их количество и общий объем программы. Для этого С/С++ представляют определенные возможности. Такие как:

1. Использование параметров-многомерных массивов.
2. Применение параметров-строк.
3. Использование параметров-функций (указателя на функцию).

#### Применение указателя на функцию как параметра.

Как уже отмечалось ранее, функция характеризуется типом возвращаемого значения, именем и сигнатурой. Имя функции – это указатель, хранящий ее адрес, который может быть присвоен другому указателю. Однако при объявлении для нового указателя должен быть задан тот же тип, что и возвращаемое функцией значение, и тот же список параметров с точностью до типов формальных параметров (имена параметров могут различаться).

Указатель на функцию объявляется следующим образом:

**<Тип функции>(\* <Имя>)(<Спецификация параметров>);**

Например:

int (\*ptrfun)(int, int);

При объявлении указатель на функцию может быть инициализирован, но в качестве значения должен быть указан адрес функции, тип и сигнатура которой соответствуют определяемому указателю, например:

Описание функций:

char f1(char){…}

char f2(int){…}

void f3(float){…}

int f4(float){…}

int f5(int){…}

Объявление указателей на функции:

void (\*ptr1)(float) = f3; // инициализированныйуказатель

int (\*ptr2)(int);

char (\*ptr3)(int);

Присваивание указателей:

ptr2 = f5; ptr3 = f2;// корректное

prt2 = f4; ptr3 = f1; // некорректное: несоответствие типов или спецификаций

**Пример 2.** Разработать подпрограммы выполнения элементарных операций.

#include <locale.h>

#include <stdio.h>

#include <conio.h>

int add(int n,int m) { return n+m; }

int sub(int n,int m) { return n-m; }

int mul(int n,int m) { return n\*m; }

int div(int n,int m) { return n/m; }

int main()

{

setlocale(0, "russian");

int (\*ptr)(int, int); // указатель на функцию

int a = 6, b = 2; char c = '+';

while (c != ' '){

printf("%d%c%d=", a, c, b);

switch (c)

{

case '+': ptr = add; c = '-'; break;

case '-': ptr = sub; c = '\*';break;

case '\*': ptr = mul; c = '/';break;

case '/': ptr = div; c = ' ';

}

printf("%d\n", a = ptr(a,b)); // вызов функции по указателю

}

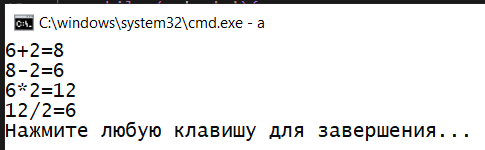
puts("Нажмите любую клавишу для завершения...");

\_getch();

return 0;

}

**Результат работы программы:**



**Пример 3.** Написать подпрограмму вычисления значения интеграла произвольных функций одной переменной на отрезке [a, b] с точностью eps.

#include <locale.h>

#include <stdio.h>

#include <conio.h>

#include <math.h>

float (\* funuk)(float); // указатель на функцию

// Функция расчета интеграла методом прямоугольников

float integral(float(\*funuk)(float), float a, float b, float eps)

{

int i, n = 5, k = 0; float s1, s2 = 1.0e+10, x, d;

d = (b-a)/n;

do

{

s1 = s2; s2 = 0; n = n\*2; d = d/2;

x = a; k++;

for(i = 1; i<=n; i++)

{

s2 = s2+funuk(x);

x = x+d;

}

s2 = s2\*d;

}

while(fabs(s2-s1)>eps);

return s2;

}

// Описание функций, для которых считается интеграл

float f1(float x) { return x\*x-1; }

float f2(float x) { return 2\*x; }

int main(int argc, char\* argv[])

{

setlocale(0, "russian");

float a, b, eps;

puts("Введите a, b, eps для функции y=x^2-1:");

scanf("%f %f %f", &a, &b, &eps);

**//вызов функции расчета интеграла для функции f1**

printf("Значение интеграла=%10.5f\n", integral(f1, a, b, eps));

puts("Введите a, b, eps для функции y=2\*x:");

scanf("%f %f %f", &a, &b, &eps);

**// вызов функции расчета интеграла для функции f2**

printf("Значение интеграла=%10.5f\n", integral(f2, a, b, eps));

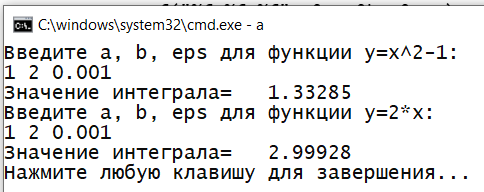
puts("Нажмите любую клавишу для завершения...");

\_getch();

return 0;

}

**Результаты работы программы:**



#### Пример использования указателя на функцию.

Пусть необходимо написать подпрограмму, которая выводила бы таблицу значений любой вещественной функции одной переменной на заданном отрезке с заданным шагом.

Так как функции могут быть разными, чтобы подпрограмма стала универсальной, в нее нужно передавать имя той функции, для которой строится таблица. Для этого целесообразно применить процедурный тип, чтобы передать в подпрограмму имя нужной функции через параметр этого типа. Схемы алгоритмов программы и подпрограммы приведены на рисунках 2 и 3.



**Рисунок 2** – Схема алгоритма подпрограммы табуляции функции

***Текст модуля с подпрограммой***

void TablicaF(float (\*funuk)(float), float a, float b, float h,char s[])

{ float fx, x;

x = a;

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

puts("| Таблица значений функции |");

printf("| %10s |\n",s);

printf("| на отрезке %5.2f -%5.2f с шагом %5.3f |\n",a,b,h);

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

puts("| X | Y |");

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

while(x<=b)

{ fx = funuk(x);

printf("| %6.2f | %8.4f |\n",x,fx);

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

x = x+h;

}

};

***Текст модуля списка функций***

#include <math.h>

float F1(float x) { return x\*x\*sin(x)-x+1; }

float F2(float x) { return x\*x-1; }

float F3(float x) { return x-3; }



**Рисунок 3** – Схема алгоритма основной программы

***Текст основной программы***

#include <locale.h>

#include "SpisFun.h"

#include "ModTabul.h"

float (\*ptr)(float);

void main(int argc, char\* argv[])

{ float xn, xk, h;

setlocale(0, "russian");

puts("Введите Xn, Xk, h for y=x^2\*sin(x)-x+1");

scanf("%f %f %f", &xn, &xk, &h);

TablicaF(F1, xn, xk, h, "y=x^2\*sin(x)-x+1");

puts("Введите Xn, Xk, h for y=x^2-1");

scanf("%f %f %f", &xn, &xk, &h);

TablicaF(F2, xn, xk, h, "y=x^2-1");

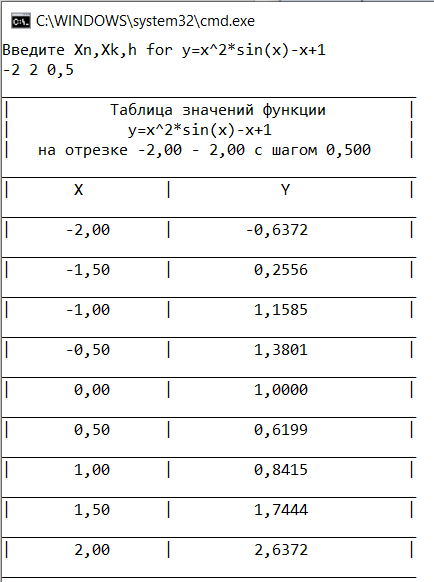
puts("Введите Xn, Xk, eps for y=x-3");

scanf("%f %f %f", &xn, &xk, &h);

TablicaF(F3, xn, xk, h, "y=x-3");

}

**Результат работы. Фрагмент.**



## Обработка списковых структур данных

*Динамическими* принято называть структуры данных, которые создаются в процессе выполнения программы. Понятие «создаются» можно интерпретировать по-разному:

* создаются – в смысле «получают память и начинают реально существовать»;
* создаются – в смысле «организуются, строятся из некоторых элементов».

К первому типу относятся динамические массивы, строки и структуры, память под которые выделяется во время выполнения программы. К структурам второго типа относятся списки.

***Список*** – структура, при организации которой использованы указатели, содержащие адреса следующих элементов. Элемент списка состоит из двух частей: информационной и адресной. *Информационная часть* содержит поля данных. *Адресная* включает от одного до n указателей, содержащих адреса других элементов. Количество связей, между соседними элементами списка определяет его *связность*: односвязные, двусвязные, n-связные.

По *структуре* списки бывают линейными, древовидными и сетевыми. На линейных списках обычно реализуют разные дисциплины обслуживания, определяющие способ формирования и обработки линейных списков:

* очередь – дисциплина обработки элементов данных, в которой добавление элементов выполняется в конец, а удаление – из начала;
* стек – дисциплина обработки элементов данных, в которой добавление и удаление элементов осуществляется с одной стороны;
* дек – дисциплина обработки элементов данных, в которой добавление и удаление элементов может выполняться с двух сторон.
* кольцевой список

Линейные односвязные списки используют чаще других списковых структур, так как они сравнительно просты, но одновременно, в отличие от одномерных массивов, позволяют:

* работать с произвольным количеством элементов, добавляя и удаляя их по мере необходимости;
* осуществлять вставку и удаление записей, не перемещая остальных элементов последовательности.

Недостатком этой структуры является то, что при поиске элемента по номеру приходится просматривать все ранее расположенные элементы, в то время как в одномерном массиве возможен прямой доступ к элементу по индексу. К тому же реализация линейного односвязного списка требует дополнительной памяти для хранения адресной части элементов.

Рассмотрим более подробно, как выполняются основные операции с линейными односвязными списками.

Исходные установки. В начале программы необходимо описать элемент и его тип:

Элемент односвязного списка:

struct element // тип элемента

{

char name[16]; // информационное поле 1

char telefon[7]; // информационное поле 2

element \*p; // адрес следующего элемента

};

Элемент двусвязного списка:

struct element // тип элемента

{

char name[16]; // информационное поле 1

char telefon[7]; // информационное поле 2

element \*prev; // адресное поле «предыдущий»

element \*next; // адресное поле «следующий»

};

В статической памяти описываем переменную-указатель списка и несколько переменных-указателей, используемых при выполнении операций со списком:

element \*first, // адрес первого элемента

\*n, \*f, \*q; // вспомогательные указатели

Исходное состояние «список пуст»:

first = NULL;

Обработку списков рассмотрим на примере добавления нового элемента к списку.

Добавление элемента к списку включает запрос памяти для размещения элемента и заполнение его информационной части. Построенный таким образом элемент добавляется к уже существующей части списка.

В общем случае при добавлении элемента к списку возможны следующие варианты:

* список пуст, добавляемый элемент станет единственным элементом списка;
* элемент необходимо вставить перед первым элементом списка;
* элемент необходимо вставить перед заданным (не первым) элементом списка;
* элемент необходимо вставить после заданного (не последнего) элемента;
* элемент необходимо дописать в конец списка.

*Добавление элемента к пустому списку* состоит из записи адреса элемента в указатель списка, причем в поле «адрес следующего» добавляемого элемента необходимо поместить NULL:

first = new element; // запросили память под элемент

first->num = 5; // занесли данные в информационное поле

first->p = NULL; // записали признак конца списка NULL

На рис. 4 показана последовательность операций при добавлении элемента к пустому списку.



**Рисунок 4 –** Последовательность операций при добавлении элемента к пустому списку: исходное состояние (*а*), запрос памяти под элемент (*б*), заполнение элемента (*в*), занесение nil в поле адреса следующего элемента (*г*)

*Добавление элемента перед первым элементом списка.* При выполнении этой операции необходимо в поле «адрес следующего» переписать адрес первого элемента списка, а в указатель списка занести адрес добавляемого элемента (рис. 5):

q = new element; // запросили память под элемент

q->num = 4; // занесли данные в информационное поле

q->p = first; // записали в новый элемент адрес первого

first = q; // записали в качестве первого адрес нового элемента



**Рисунок 5 –** Последовательность операций при добавлении элемента перед первым: исходное состояние (*а*), запрос памяти под элемент (*б*), заполнение элемента (*в*), шаги включения элемента в список (*г*, *д*)

*Добавление элемента перед заданным (не первым).*Для выполнения операции необходимо знать адреса элементов, между которыми вставляется элемент, так как адресные части этих элементов при выполнении операции будут корректироваться (рис. 6). Пусть f – адрес предыдущего элемента, а n – адрес следующего элемента, тогда:

q = new element; // запросили память под элемент

q->num = 4; // занесли данные в информационное поле

q->p = n; // в поле «адрес следующего» нового элемента переписываем

//адрес следующего элемента

f->p = q; // в поле «адрес следующего» предыдущего элемента

//заносим адрес нового элемента



**Рисунок 6 –** Добавление элемента перед заданным (не первым): исходное состояние (*а*), запрос памяти под элемент (*б*), заполнение элемента (*в*), шаги включение элемента в список (*г*, *д*)

*Добавление элемента в конец списка.* В этом случае должен быть известен адрес элемента (f), после которого добавляется новый элемент (рис. 7):

q = new element; // запросили память под элемент

q->num = 4; // занесли данные в информационное поле

q->p = NULL; // записали признак конца списка NULL

f->p = q; // в поле «адрес следующего» предыдущего элемента

//заносим адрес нового элемента



**Рисунок 7 –** Добавление элемента в конец списка: исходное состояние (*а*), запрос памяти под элемент (*б*), заполнение элемента (*в*), включение элемента в список (*г*, *д*)

При работе со списками, многие из приемов их обработки можно оформлять в виде подпрограмм. При этом, указатели на элементы списков ( на начала, на конец и т.д.) могут передаваться, как параметры. В этом случае сами параметры являются указателями. Мы знаем, что в С/С++ все параметры по умолчанию передаются по значению. Это касается и самих указателей. Они тоже передаются по значению. А это значит, что в стеке создается копия указателя и все изменения с такими параметрами в вызывающую программу не передаются. Для подпрограмм печати или каких-то подсчетов это вполне применимо. Однако, если в подпрограмме меняется значение самого указателя, то необходимо использовать адреса этих указателей, то есть использовать указатель или ссылку. Оба эти способы используются, но применение указателя на указатель (адреса указателя) более предпочтительно. Например:

// Описание элемента списка целое число

struct rec {

int value;

rec \*p;

};

*Заголовок функции добавления*

void add\_element (rec \*\*first,rec \*\*last, int val);

rec \*begin, \*end,\*next,\*q;// описание переменных для работы со списком

*Вызов подпрограммы*

add\_element (&begin,&end,a);

Более подробно особенности выполнения операций со списками рассмотрены в [1].

**Пример.** Написать программу, которая вводит символы из символьной строки, содержащей несколько символов ‘\*’ (“qwe\*\*\*asd\*sdf\*\*\*”) и строит из символов строки список-очередь. Выводит построенный список. В полученный список перед каждым символом ‘\*’ вставляет символ ‘#’ (“qwe#\*#\*#\*asd#\*sdf#\*#\*#\*). Полученный список выводит на экран.

Текст программы приведен ниже:

#include "stdafx.h"

#include <stdio.h>

#include <string.h>

// Описание элемента списка

struct rec {

char ch;

rec \*p;

};

void main() {

rec \*f, \*first, \*q, \*p, \*last;

char st[40];

puts("Input string with \*");

gets(st);

// Формирование списка

first = new rec;

first->p = NULL;

first->ch = st[0];

last = first;

int i = 1;

while(st[i] != '\0')

{

q = new rec;

q->ch = st[i];

q->p = NULL;

last->p = q;

last = q;

i++;

}

// Печать сформированного списка

puts("Inputed spisok");

q = first;

while(q != NULL)

{

printf("%2c", q->ch);

q = q->p;

}

printf("\n");

// Проверка первого элемента списка на равенство '\*' и добавление '#'

if(first->ch=='\*')

{

q = new rec;

q->ch = '#';

q->p = first;

first = q;

f = first->p;

q = f;

}

else

q = f = first;

//Проверка остальных элементов списка на равенство '\*' и добавление'#'

while(q->p != NULL)

{

f = q;

q = q->p;

if (q->ch == '\*')

{

p = new rec;

p->ch = '#';

p->p = q;

f->p = p;

}

}

// Печать скорректированного списка

puts("NEW spisok");

q = first;

while(q!=NULL)

{

printf("%2c",q->ch);

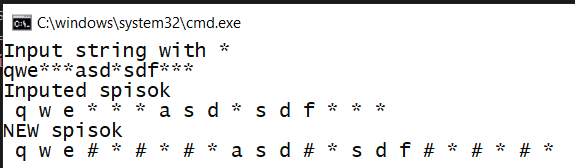
q=q->p;

}

printf("\n");

}

**Результат работы программы:**



## Тестирование

Тестирование – очень важный и трудоемкий этап процесса разработки программного обеспечения, так как правильное тестирование позволяет выявить подавляющее большинство ошибок, допущенных при составлении программ.

Для тестирования создано множество специальных методик, рассматривающих программу под разными углами и позволяющих выявить разные виды ошибок, но в этом разделе будут рассмотрены самые базовые и обобщенные принципы тестирования.

В основе *структурного тестирования* лежит концепция максимально полного тестирования всех маршрутов (т.е. последовательностей операторов) программы. Так, если алгоритм программы включает ветвление, то при одном наборе исходных данных может быть выполнена последовательность операторов, реализующая действия, которые предусматривает одна ветвь, а при втором – другая. Соответственно для программы будут существовать маршруты, различающиеся выбранным при ветвлении вариантом.

Формирование тестовых наборов для тестирования маршрутов может осуществляться по нескольким критериям, например, чтобы каждый оператор программы выполнялся по крайней мере один раз, чтобы были выполнены все возможные результаты каждого условия в программе и так далее.

Еще одним из способов проверки программ является тестирование по принципу *«черного ящика»*. В этом случае программа рассматривается как «черный ящик», и целью тестирования является выяснение обстоятельств, в которых поведение программы не соответствует требованиям.

Один из методов предполагает разбиение всех возможных значений каждого входного параметра на конечное число групп. Например, если в условии задачи сказано, что , то можно выделить одну правильную область значений и две неправильные: и .

Таким образом для каждого параметра выделяются области правильных и неправильных значений, а затем для каждой неправильной области формируют свой тест. Например, у программы есть два входных параметра – целые числа и , при этом . Можно выделить следующие области значений:

* неправильные: для *x* и для *y*.
* правильные: для *x* и для *y*.

Тестовые наборы могут быть следующими:

1. (неправильное значение , правильное )
2. (неправильное значение , правильное )
3. (правильное *,* неправильное значение )
4. (правильные значения и для , и для )

Еще один метод выделения тестовых наборов данных – *анализ граничных значений*. Из названия следует, что данный метод проверяет значения на границах правильных и неправильных областей значений и в их окрестностях, например, для целочисленного должны быть сгенерированы тесты для и .

**Тестирование и отладка программ, работающих со списковыми структурами**

При тестировании работы со списками можно применять методы, описанные ранее. Например, если в условии задачи заданы ограничения для информационного поля, то необходимо сгенерировать тесты, в которых

1. не будет ни одного подходящего по условию элемента;
2. все элементы удовлетворяют условию;
3. часть значений подходят, а часть – нет;
4. список пуст (не введены входные параметры).

Также нужно сформировать другие виды тестов, основанные на значениях входных данных, зависящие от условий конкретной задачи. Необходимо проверить все реализованные в программе функции, например, удаление и добавление элементов, сортировка и т.д. в зависимости от выполняемой задачи.

Для отладки программы необходимо подобрать входные данные, при которых возникает ошибка, а затем, пошагово выполняя программу, особенно внимательно отслеживать значения указателей первого, текущего, следующего элементов, содержимое адресных полей элементов списка. Необходимо учитывать, чтобы содержимое адресных полей элемента было корректным, так как часто встречаются ошибки с так называемыми *висячими указателями*, когда адресное поле содержит указатель на элемент, память из-под которого уже освободили. Также следует избегать *утечек памяти*, когда динамическая память вовремя не освобождается должным образом и остается зарезервирована, хотя, фактически, программа уже не может получить доступ к этой области памяти. Например, рассмотрим следующий фрагмент программы:

char \*pointer = NULL;

for( int i = 0; i < 10; i++ ) {

pointer = new char[100];

}

delete [] pointer;

В этом примере в 3-й строке создается объект в динамической памяти. Оператор на 3-й строке выполняется 10 раз, причём каждый следующий раз адрес нового объекта перезаписывает значение, хранящееся в указателе pointer. На 5-й строке выполняется удаление объекта, созданного на последней итерации цикла. Однако первые 9 объектов остаются в динамической памяти, и одновременно в программе не остается переменных, которые бы хранили адреса этих объектов. То есть в 5-й строке невозможно ни получить доступ к первым 9 объектам, ни удалить их.

# Практическая часть.

## Порядок выполнения задания

Домашнее задание №3 выдается на 11-й неделе и включает разработку алгоритмов программ и разработку текстов программ задания 3. При этом допускается использовать как схемы алгоритма, так и псевдокоды. Все алгоритмы должны быть структурными, т. е. должны использовать только допустимые управляющие структуры.

Домашнее задание состоит из двух задач. Первая задача на применение средств разработки универсальных программ, в частности, указателя на функцию. Вторая задача на программирование формирования и обработки списковых структур данных.

Для выполнения каждой из задач необходимо:

1. Прочитать и проанализировать задание в соответствии со своим вариантом.
2. Разработать алгоритм программы.
3. Разработать тесты для тестирования и отладки программы в соответствии с ее особенностями.
4. В среде программирования Visual Studio 2008 или Visual Studio 2017 Community создать новый проект и закодировать программу в соответствии с разработанным алгоритмом.
5. Отладить программу.
6. Составить отчет по домашнему заданию.
7. Продемонстрировать работу программы преподавателю.
8. Защитить задание преподавателю.

## Требования к оформлению.

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Если задания сдаются поэтапно (каждая часть отдельно), то для каждой части оформляется отдельный отчет.

Отчет по каждой части ДЗ должен содержать:

1. Титульный лист (Пример приведен в приложении А).
2. Текст задания.
3. Схему алгоритма.
4. Текст программы.
5. Пример работы программы.
6. Таблицу тестовых примеров или скриншоты работ программы.
7. Выводы.
8. Ответы на контрольные вопросы. Не менее 3 ответов.

Однако, если задания ДЗ оформляются одним отчетом, то титульный лист может быть один на все части, а ответы на обе части размещены в конце отчета ( не менее 6 ответов на все ДЗ). Остальные пункты требований к оформлению должны быть для каждого задания.

## Требования к защите

Для получения зачета по домашнему заданию, студент должен представить отчет, оформленный в соответствии с требованиями, продемонстрировать работающие программы и ответить на вопросы преподавателя по программе и теории.

Проверка домашнего задания может осуществляться поэтапно, по мере выполнения практических заданий. Окончательный срок завершения задания – 16 неделя 1-го семестра.

Студент считается выполнившим домашнее задание, если у него к этому моменту завершены, проверены преподавателем отчеты по выполнению указанных заданий и задания защищены.

## Пример выполнения задания

**Часть 1. Создание модулей. Указатель на функцию.**

**Задание.** Разработать модуль, содержащий указанную процедуру или функцию. Написать тестирующую программу.

Составить подпрограмму для определения значения многочлена .

В подпрограмму передается вектор коэффициентов и значение переменной , а возвращается значение многочлена. Составить подпрограмму Табуляции многочлена на отрезке c шагом , которые вводятся с клавиатуры.

Использовать написанную процедуру для вычисления значения многочлена при выполнении построения таблицы функций многочленов:

Ввод данных выполнить в основной программе.

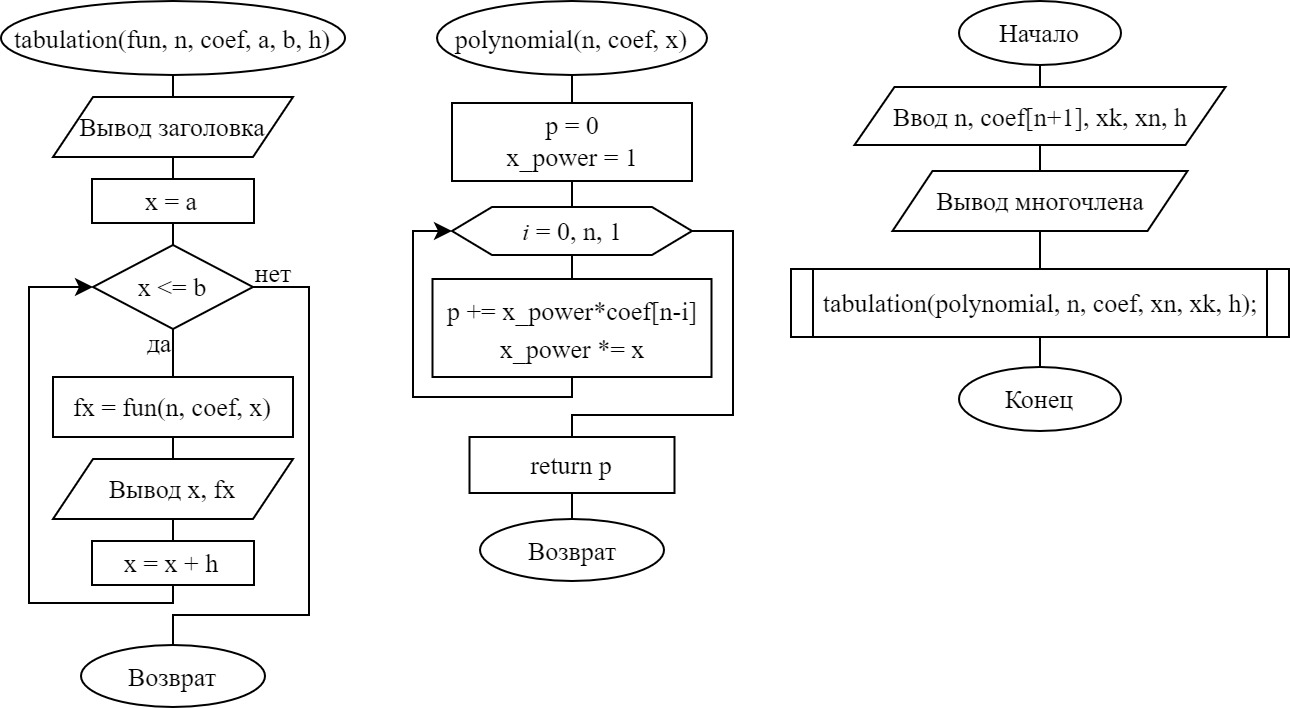
Анализ показал, что для выполнения задания необходимо разработать модуль, содержащий функцию вычисления значения степенного многочлена, входными параметрами которой будут массив коэффициентов и значение переменной *x*, а выходным – посчитанное значение многочлена. Также необходимо создать модуль, реализующий подпрограмму табуляции, основу которой можно взять в разделе 1.1.3.2.

Структурная схема программы приведена на рисунке 8.



**Рисунок 8.** Структурная схема программы

Схемы алгоритмов для всех разрабатываемых подпрограмм и основной программы показаны на рисунке 9.



**Рисунок 9** – Схемы алгоритмов

Разработаем тексты программ согласно этим схемам.

**Файл polynomial.cpp:**

#include "polynomial.h"

#include <stdio.h>

double polynomial(int n, double\* coef, double x) {

double p = 0, x\_power = 1;

for (int i = 0; i <= n; i++) {

p += x\_power\*coef[n-i];

x\_power \*= x;

}

return p;

}

**Файл tabulation.cpp:**

#include <stdio.h>

#include "tabulation.h"

void tabulation(double (\*fun)(int, double\*, double), int n, double\* coef, double a, double b, double h) {

double fx, x;

x = a;

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

puts("| Таблица значений многочлена |");

printf("| на отрезке %5.2lf - %5.2lf с шагом %5.3lf |\n", a, b, h);

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

puts("| X | Y |");

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

while(x <= b)

{

fx = fun(n, coef, x);

printf("| %6.2lf | %15.4lf |\n", x, fx);

puts("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

x = x + h;

}

};

**Файл main.cpp:**

#include <locale.h>

#include "polynomial.h"

#include "tabulation.h"

int main(int argc, char\* argv[]) {

double xn, xk, h, \*coef;

int n;

setlocale(0, "russian");

puts("Введите степень многочлена:");

scanf("%d", &n);

coef = new double[n+1];

printf("Введите %d коэффициентов многочлена, начиная с коэффициента старшей степени:\n", n+1);

for (int i = 0; i <= n; i++) scanf("%lf", &coef[i]);

puts("Введите Xn, Xk, h для вычисляемой функции:");

scanf("%lf %lf %lf", &xn, &xk, &h);

printf("\n");

puts("Введенный многочлен:");

printf("P(x) = ");

for (int i = 0; i < n; i++)

printf("(%.2lf)\*x^%d + ", coef[i], n-i);

printf("(%.2lf)\n", coef[n]);

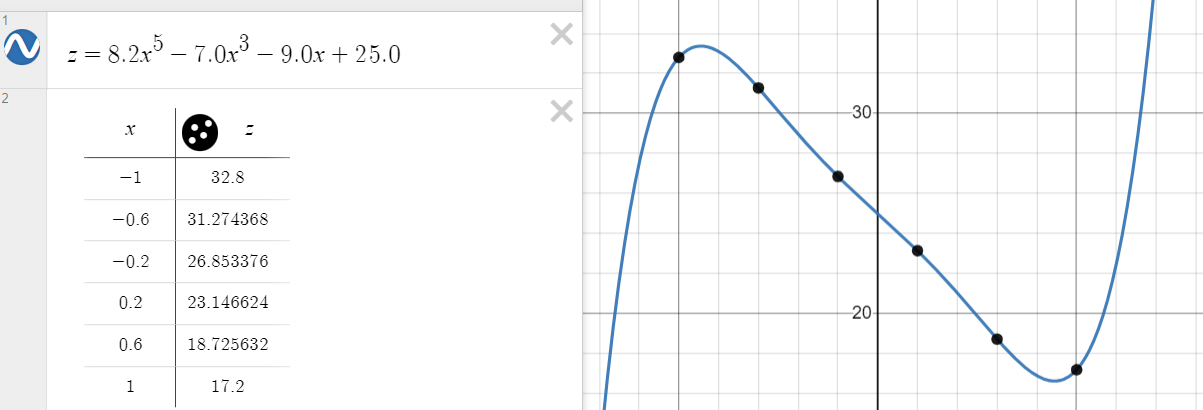
tabulation(polynomial, n, coef, xn, xk, h);

delete[] coef;

return 0;

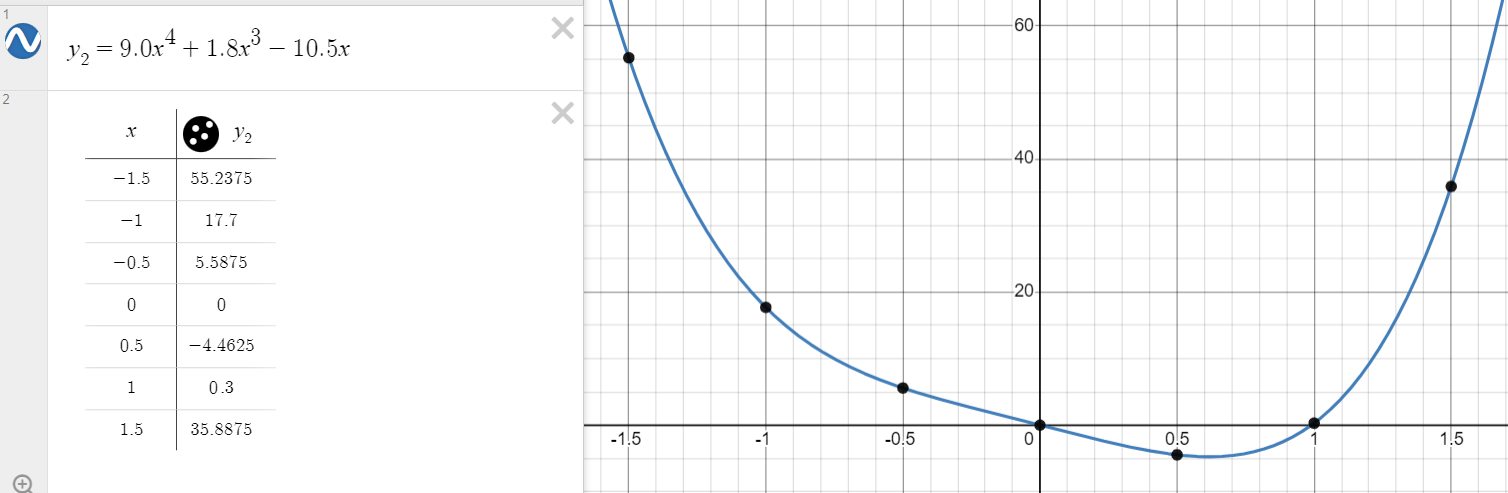
}

Аналитически вычислим значения многочленов, указанных в условии. Рассмотрим значения первого многочлена на отрезке с шагом 0.4, результаты представлены на рисунке 10.

****

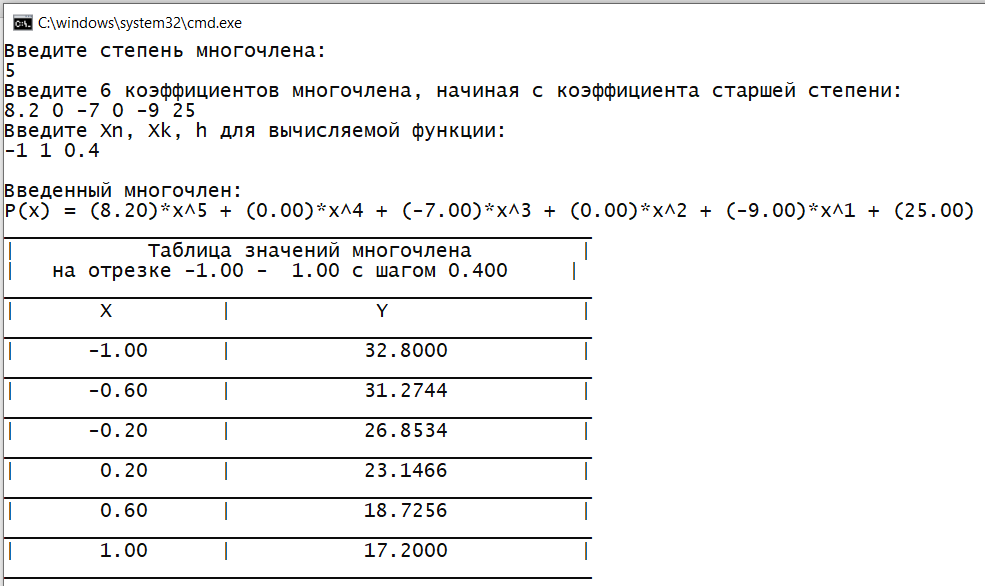
**Рисунок 10** – Расчетные значения первого многочлена

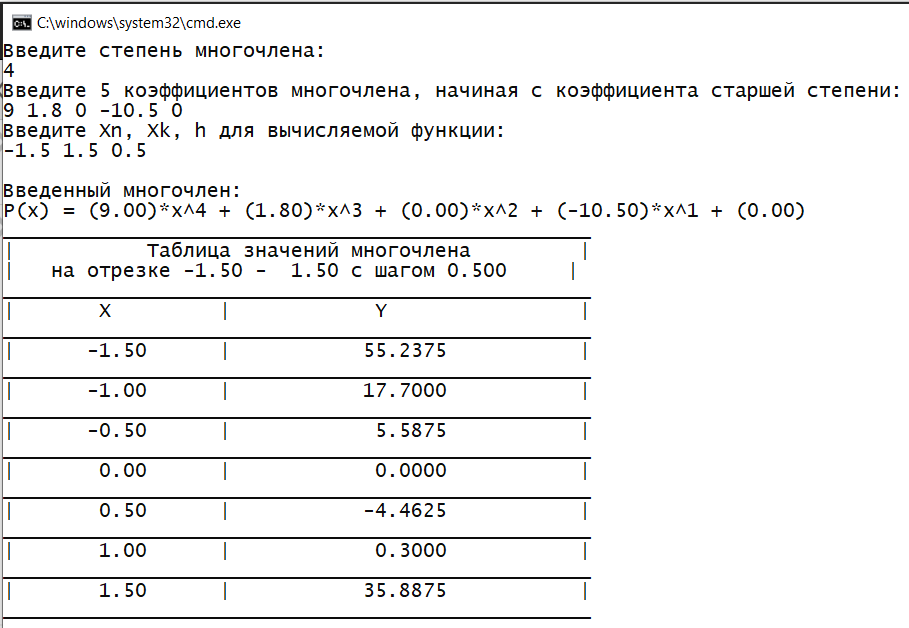
Рассмотрим значения второго многочленана отрезке с шагом 3, результаты представлены на рисунке 11.



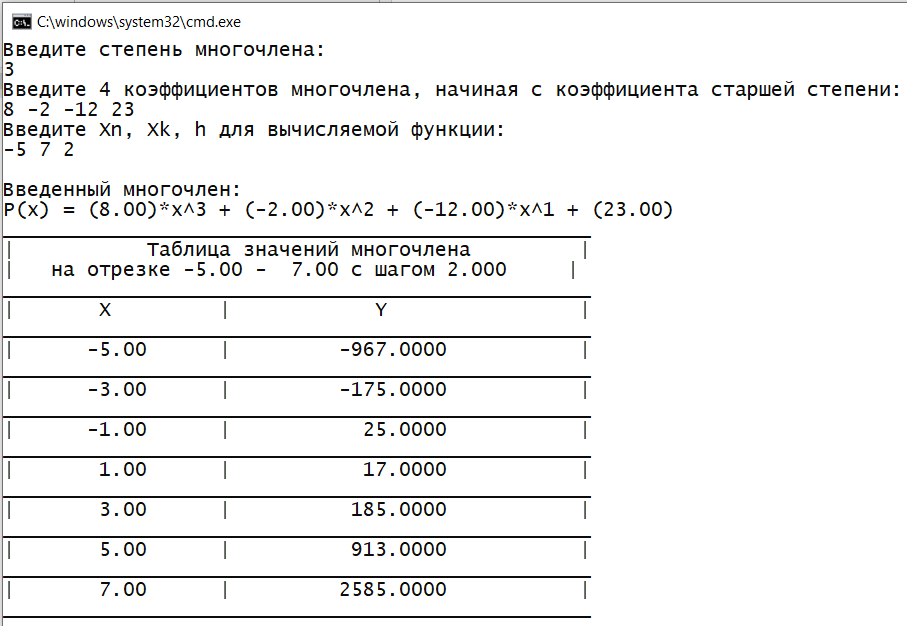
**Рисунок 11** – Расчетные значения второго многочлена

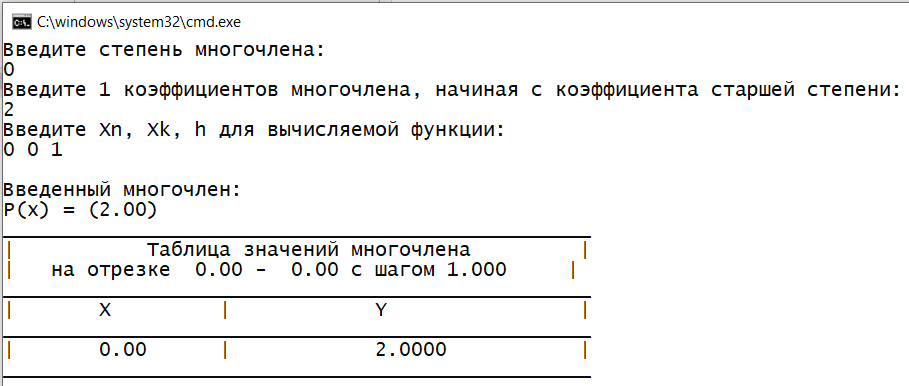
**Результат работы программы:**

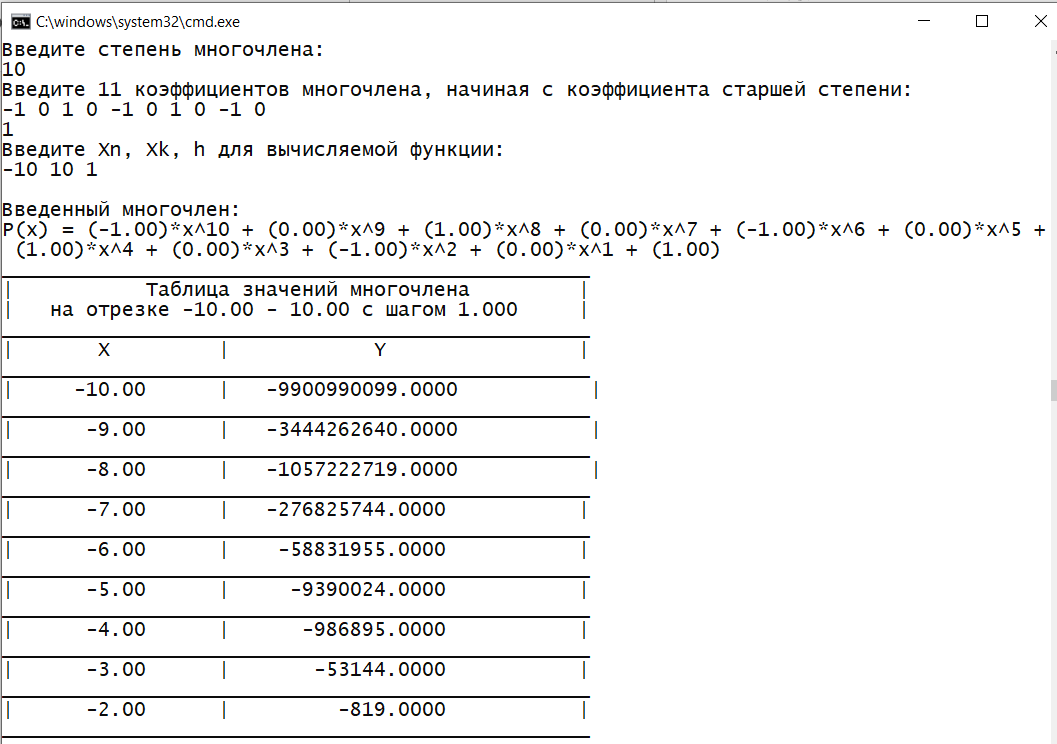




**Тесты:**

****





**Часть 2. Строки и динамические структуры.**

**Задание**. С клавиатуры вводятся строки, содержащие целые числа, записанные через пробел. Организовать из вводимых чисел однонаправленный список, упорядоченный по абсолютной величине числа. В программе предусмотреть возможность добавления в список нового элемента и удаления элемента, равного введенному с клавиатуры. Вывести на печать список и проверить функции добавления и удаления элементов списка.

Как показал анализ задания, необходимо организовать работу со списком целых чисел. Для этого можно выделить подпрограммы работы со списком в отдельный модуль. Упорядочивание чисел в списке по абсолютной величине можно реализовать двумя способами: 1) при добавлении элемента сразу вставлять его в нужное место; 2) добавлять новый элемент в конец, а затем сортировать список. Так как второй способ является намного менее эффективным, будем реализовывать первый. Таким образом, необходимо реализовать

* ввод списка,
* печать списка,
* добавление нового элемента,
* удаление элементов по значению.

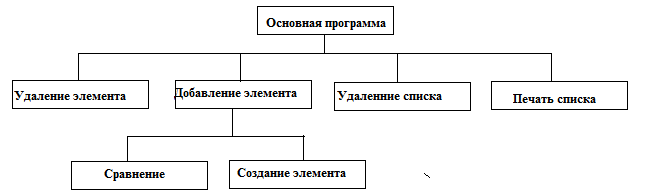
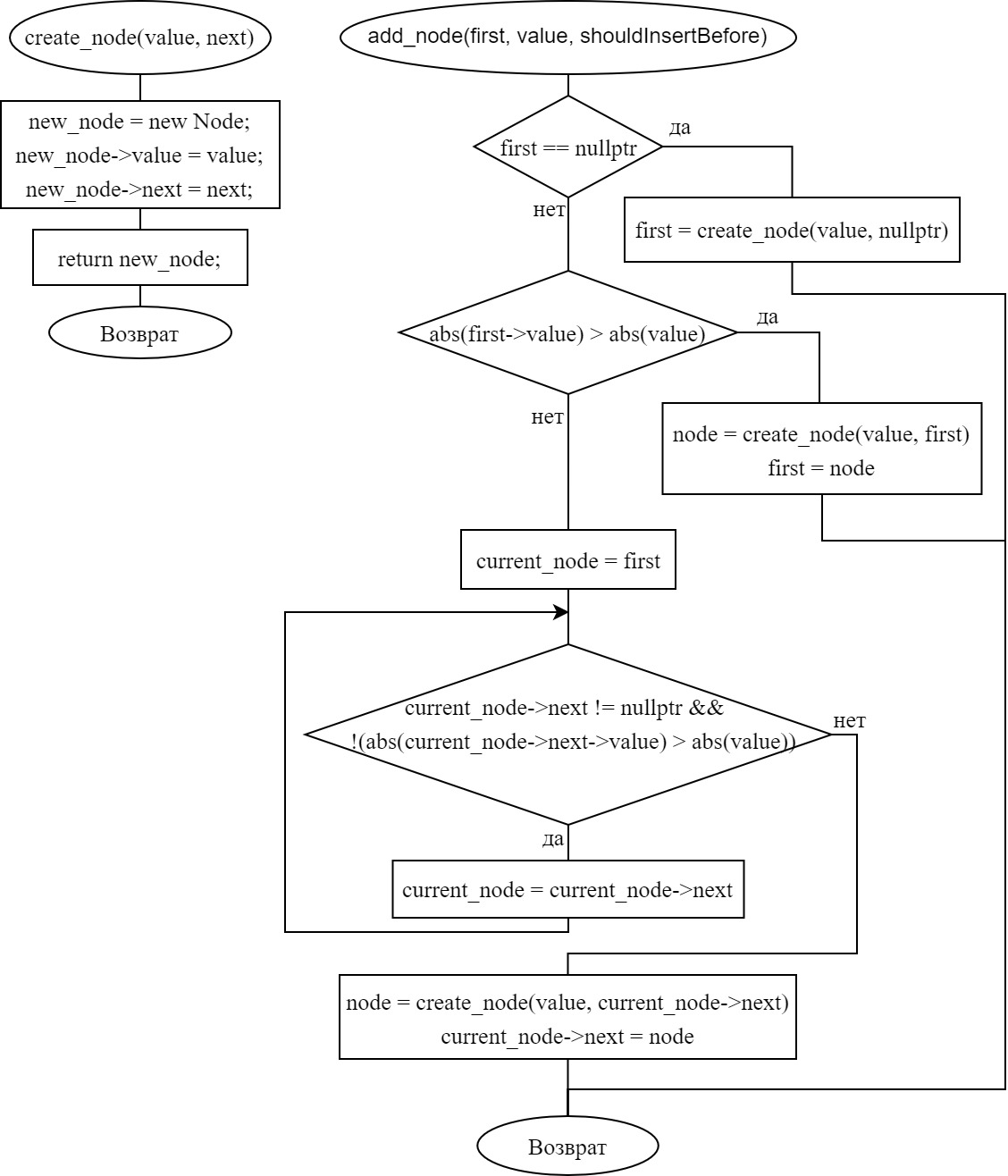
Структурная схема программы приведена на рисунке 12.

Рисунок 12. Структурная схема программы

Спроектируем схемы алгоритмов для разрабатываемых подпрограмм и основной программы, они показаны на рисунках 13-15.



**Рисунок 13** – Схемы алгоритмов подпрограмм add\_node и create\_node

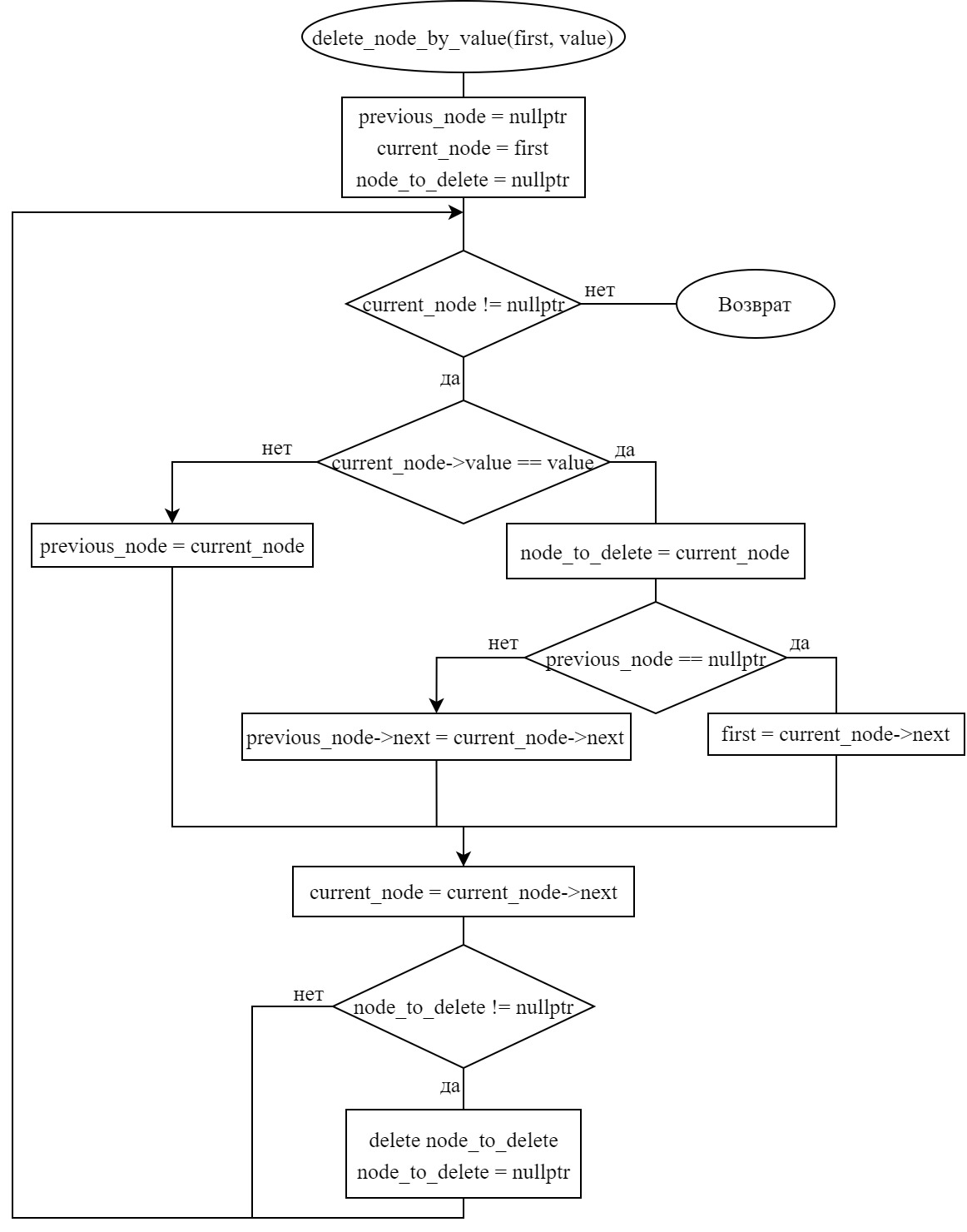


Рисунок 14 – Схема алгоритма подпрограммы delete\_node\_by\_value

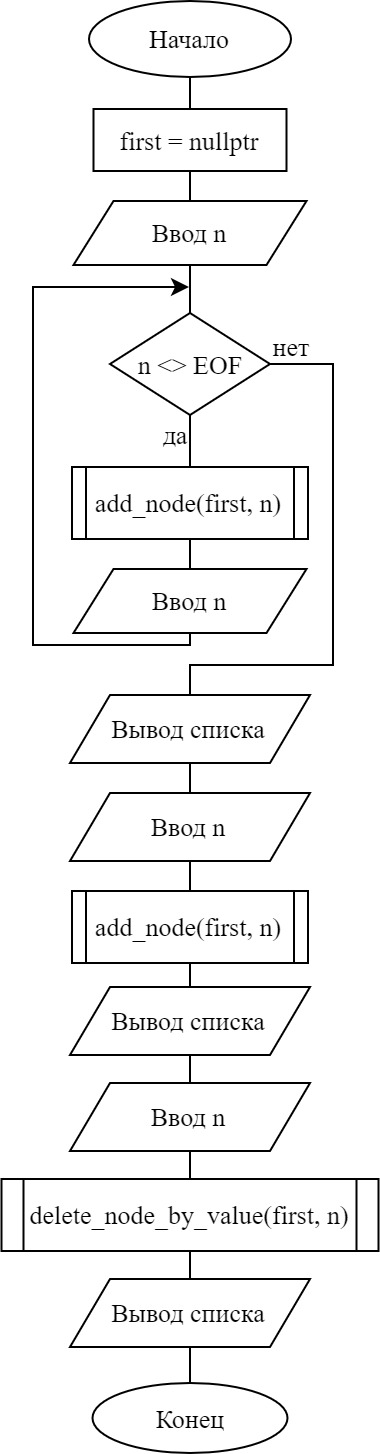


Рисунок 15 – Схема алгоритма основной программы

Часть наиболее простых схем алгоритмов подпрограмм опущены. Однако в отчет необходимо включить схемы алгоритмов всех подпрограмм.

Разработаем текст программы согласно этим схемам алгоритмов. Однако, для большей универсальности подпрограммы добавления элемента в список было решено заменить исходное условие «abs(value1)>abs(value2)» на функцию shouldInsertBefore(int a, int b), которая возвращает результат сравнения (или другого логического выражения) над параметрами a и b в виде логического значения, на основе которого принимается решение о том, куда именно вставлять новый элемент списка. Для реализации в функции add\_node будем использовать указатель на функцию, а сам закон упорядочивания в виде функции передавать в качестве аргумента при вызове.

**Текст программы**:

**Файл list.h:**

#ifndef listh\_20230721

#define listh\_20230721

struct Node {

int value;

Node \* next;

};

// введем более удобное обозначение для указателя на элемент

using NodePtr = Node\*;

NodePtr create\_node(int value, NodePtr next);

void add\_node(NodePtr &first, int value,

bool (\*shouldInsertBefore)(int, int));

// shouldInsertBefore(node->value, value) == true <=> вместо node

// вставляем новое значение

void delete\_node\_by\_value(NodePtr &first, int value);

#endif

**Файл list.cpp:**

#include "list.h"

NodePtr create\_node(int value, NodePtr next) {

NodePtr new\_node = new Node;

new\_node->value = value;

new\_node->next = next;

return new\_node;

}

void add\_node(NodePtr &first, int value,

bool (\*shouldInsertBefore)(int, int)) {

// добавляем первый элемент

if (first == nullptr) {

first = create\_node(value, nullptr);

return;

}

// если новый элемент должен располагаться перед первым

if (shouldInsertBefore(first->value, value)) {

NodePtr node = create\_node(value, first);

first = node;

return;

}

NodePtr current\_node = first;

// передвигаемся по списку, пока не найдем место, куда нужно вставить элемент

while (current\_node->next != nullptr

&& !shouldInsertBefore(current\_node->next->value, value))

current\_node = current\_node->next;

// добавляем элемент в список

NodePtr node = create\_node(value, current\_node->next);

current\_node->next = node;

return;

}

void delete\_node\_by\_value(NodePtr &first, int value) {

NodePtr previous\_node = nullptr, current\_node = first;

NodePtr node\_to\_delete = nullptr;

while (current\_node != nullptr) {

if (current\_node->value == value) {

// запоминаем элемент для удаления

node\_to\_delete = current\_node;

if (previous\_node == nullptr) {

// если удаляем первый элемент, сдвигаем голову списка

first = current\_node->next;

} else {

// иначе связываем предыдущий и следующий элементы

previous\_node->next = current\_node->next;

}

} else {

// сдвигаем указатель предыдущего элемента

previous\_node = current\_node;

}

// сдвигаем указатель текущего элемента

current\_node = current\_node->next;

// удаляем запомненный элемент, если такой есть

if (node\_to\_delete != nullptr) {

delete node\_to\_delete;

node\_to\_delete = nullptr;

}

}

}

**Файл main.cpp:**

#include "list.h"

#include <stdio.h>

#include <math.h>

// функция сравнения по абсолютной величине

bool compare\_func(int a, int b) {

return abs(a) > abs(b);

}

// процедура печати списка

void print\_list(NodePtr first) {

NodePtr node = first;

if (node == nullptr)

puts("The list is empty");

else {

while (node != nullptr) {

printf("%d ", node->value);

node = node->next;

}

printf("\n");

}

}

// процедура удаления списка

void delete\_list(NodePtr first) {

NodePtr node = first, node\_tmp;

while (node != nullptr) {

node\_tmp = node;

node = node->next;

delete node\_tmp;

}

}

int main(int argc, char\* argv[]) {

NodePtr first = nullptr;

int n;

puts("Input strings of integer numbers. End input by Ctrl+Z:");

while (scanf("%d", &n) != EOF)

add\_node(first, n, compare\_func);

puts("Inputed list:");

print\_list(first);

puts("Input value that you want to add to the list:");

scanf("%d", &n);

add\_node(first, n, compare\_func);

puts("List with added element:");

print\_list(first);

puts("Input value that you want to delete from the list:");

scanf("%d", &n);

delete\_node\_by\_value(first, n);

puts("List without deleted element:");

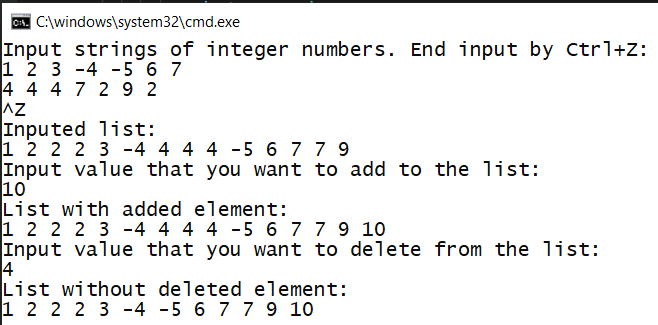
print\_list(first);

delete\_list(first);

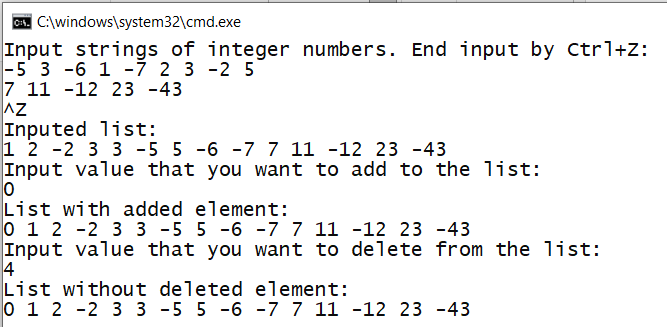
return 0;

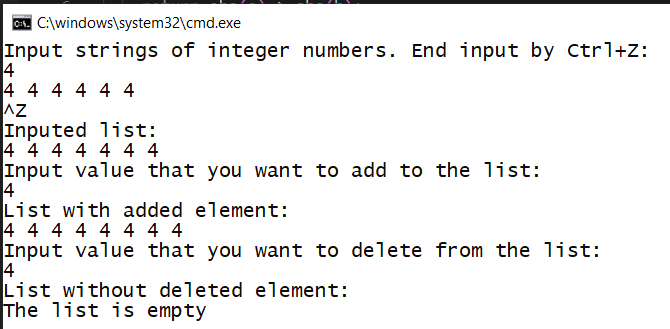
}

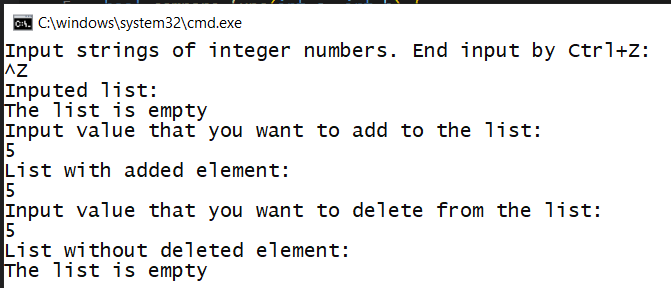
**Результат работы программы и тесты.**



**Тесты**:







## Примеры вариантов домашнего задания

**Домашнее задание 3. Вариант 27.**

**Часть 1. *Создание модулей. Указатель на функцию*.**

**Задание.** Разработать модуль, содержащий указанную процедуру или функцию. Написать тестирующую программу.

Составить подпрограмму для определения значения многочлена

. В подпрограмму передается вектор коэффициентов и значение переменной , а возвращается значение многочлена. Составить подпрограмму Табуляции многочлена на отрезке c шагом *h*, которые вводятся с клавиатуры.

Использовать написанную процедуру для вычисления значения многочлена при выполнении построения таблицы функций многочленов:

Ввод данных выполнить в основной программе.

**Часть 2. *Строки и динамические структуры.***

**Задание.** Дана символьная строка длиной не более 80 символов, состоящая из слов, разделенных пробелом. Составить программу, которая вводит строку, организует из слов строки двунаправленный список, упорядоченный по алфавиту, выводит на экран список в прямом и обратном направлении. Определить количество слов в списке, длина которых меньше 5 символов. Вывести на экран полученную информацию.

**Домашнее задание 3. Вариант 28.**

**Часть 1. Создание модулей. Указатель на функцию.**

**Задание.** Разработать модуль, содержащий указанную процедуру. Написать тестирующую программу.

Составить подпрограмму-процедуру *NEIBR* проверки принадлежности точки плоскости с координатами данной кривой .

В основной программе использовать процедуру *NEIBR* для проверки принадлежности десяти различных точек кривым и .

**Часть 2. Строки и динамические структуры.**

**Задание.** С клавиатуры вводятся строки, содержащие целые числа, записанные через пробел. Организовать из вводимых чисел однонаправленный список, упорядоченный по абсолютной величине числа. В программе предусмотреть возможность добавления в список нового элемента и удаления элемента, равного введенному с клавиатуры. Вывести на печать список и проверить функции добавления и удаления элементов списка.

# Контрольные вопросы

1. Дайте определение подпрограммы. Приведите примеры подпрограмм.
2. Что такое модуль, как его создать с помощью средств С/С++?
3. Укажите особенности подключения модулей в среде Visual Studio и Qt Creator.
4. Что такое указатель на функцию? Приведите примеры описания и применения.
5. Какие особенности передачи в подпрограмму указателей на функции как параметров? Приведите примеры из вашей программы.
6. Где и когда целесообразно использовать указатели на функции?
7. Как правильно подобрать тестовые данные для тестирования и отладки программы?
8. Что такое «список» в программировании? В каких случаях используется эта конструкция?
9. Какие типы списковых структур вы знаете?
10. Что такое указатели и как они объявляются? Приведите примеры синтаксиса и описания.
11. Как описывается элемент списка? Приведите примеры.
12. Какие варианты возможны для добавления элемента к списку?
13. Как отлаживают программы, содержащие обработку списков?
14. Какие рекомендации существуют для подбора тестовых данных списковых структур?

# Заключение

В результате выполнения домашнего задания 3 студенты получат навыки декомпозиции программы на подпрограммы, для которых выберут способ передачи параметров и средства разработки универсальных программ. Научатся оформлять и использовать выбранные средства на языке программирования С++. Кроме того, студенты получат опыт создания и обработки динамических структур данных на примере списочных структур.

В процессе кодирования и отладки студенты получат знания и умения работать в среде программирования, использовать средства отладки модульных программ, обнаруживать и исправлять ошибки. Кроме того, наличие встроенного отладчика даст возможность студентам научиться составлять тесты для обнаружения ошибок в многомодульных программах и локализовать обнаруженные при тестировании программы ошибки, используя встроенные средства отладки.

# Список литературы

Г.С. Иванова, Т.Н. Ничушкина, Р.С. Самарев. Средства процедурного программирования Microsoft Visual С++ 2008. Учебное пособие. – М.: Издательство МГТУ им. Н.Э. Баумана, 2011.

Г.С. Иванова. Программирование. Учебник для ВУЗов. – М.: Кнорус, 2013.- 432 с.

Г.С. Иванова. Технология программирования. Учебник для ВУЗов. – М.: Кнорус, 2013. – 336 с.

Подбельский В.В. Язык С++: Учеб. пособие. – М.: Финансы и статистика, 2006.

# Приложение. Пример титульного листа.

|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ **ФУНДАМЕНТАЛЬНЫЕ НАУКИ**

КАФЕДРА **ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА И МАТЕМАТИЧЕСКАЯ ФИЗИКА (фн11)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **02.03.01 МАТЕМАТИКА И КОМПЬЮТЕРНЫЕ НАУКИ**

**Отчет**

|  |  |
| --- | --- |
| **по домашнему заданию №** | 3 |

**Название:**

***Часть 3.1.* Создание модулей. Указатель на функцию.**

***Часть 3.2.* Строки и динамические структуры данных.**

**Вариант 12**

**Дисциплина: Информатика**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ФН11-23Б |  |  | В.П.Малышев |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | Т.Н.Ничушкина |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2023